AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Programowanie zaawansowane – P2 (projekt dwuosobowy)

Autor:
Rafał Curzydło
Mateusz Magiera

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

Głównym celem projektu jest realizacja programu konsolowego w języku C++, który implementuje strukturę binarnego drzewa poszukiwań (BST). Równie ważnym celem, stanowiącym trzon zadania, jest praktyczne przećwiczenie pracy grupowej z wykorzystaniem systemu kontroli wersji Git oraz platformy GitHub. Obejmuje to naukę zarządzania gałęziami (branching), łączenia zmian (merging) oraz, co kluczowe, aktywnego rozwiązywania konfliktów scalania.

1.1. Wymagania Funkcjonalne (Program)

Aplikacja musi być zaimplementowana w języku C++ z wykorzystaniem kontenerów STL. Logika programu powinna być podzielona na oddzielne klasy, każda zaimplementowana w osobnych plikach nagłówkowych (.h) i źródłowych (.cpp).

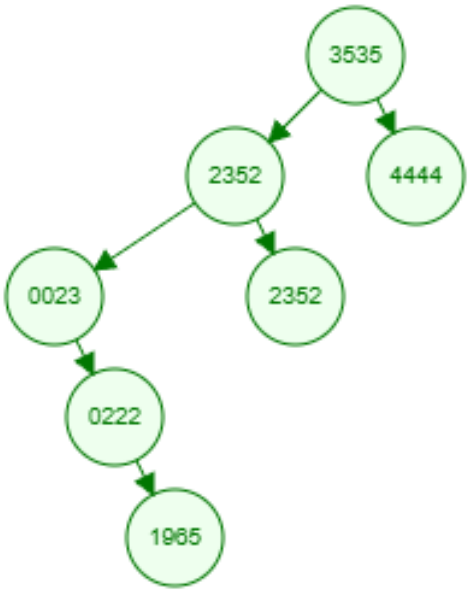
1.1.1. Klasa Drzewa (BSTree)

Pierwsza klasa, `BSTree`, powinna implementować samą strukturę drzewa BST[1]. Wymagana funkcjonalność tej klasy obejmuje następujące metody:

- Dodawanie nowego elementu (węzła) do drzewa.
- Usuwanie elementu o podanej wartości.
- Usuwanie całego drzewa (zwalnianie pamięci).
- Wyszukiwanie i wyświetlanie ścieżki do podanego elementu.
- Zapisywanie wygenerowanej struktury drzewa do pliku tekstowego.
- Wyświetlanie drzewa na ekranie, z możliwością wyboru przez użytkownika jednej z trzech metod przechodzenia: *preorder*, *inorder* oraz *postorder*.

Rysunek 1.1 (s. 5) przedstawia wizualizację koncepcji binarnego drzewa poszukiwań, gdzie węzły są dodawane zgodnie z ich wartością w stosunku do rodzica.

[1]Binarne drzewo poszukiwań (BST) to struktura danych, w której każdy węzeł ma co najwyżej dwoje dzieci, a wartość w lewym poddrzewie jest mniejsza od wartości węzła, natomiast wartość w prawym poddrzewie jest od niej większa[1].



Rys. 1.1. Wizualizacja działania binarnego drzewa poszukiwań[2]

1.1.2. Klasa Zarządzania Plikami (FileManager)

Druga klasa, `FileManager`, powinna być odpowiedzialna za zaawansowane operacje wejścia/wyjścia. Wymagane metody to:

- Zapis drzewa BST do pliku binarnego.
- Odczyt drzewa BST z pliku binarnego (odtworzenie struktury).
- Wczytanie pliku tekstowego zawierającego wyłącznie cyfry, w celu zbudowania drzewa (lub dodania elementów do istniejącego).

1.1.3. Program Główny (main.cpp)

Plik główny, zawierający funkcję `main()`, musi być odseparowany od logiki klas. Jego zadaniem jest:

- Wyświetlanie interaktywnego menu w konsoli, pozwalającego na wybór wszystkich funkcji zdefiniowanych w klasach `BSTree` i `FileManager`.
- Oczekiwanie na wybór opcji przez użytkownika.
- Zapewnienie opcji poprawnego zamknięcia programu.

1.2. Wymagania Procesowe (GitHub)

Kluczową częścią zadania jest demonstracja umiejętności pracy z systemem kontroli wersji Git w zespole dwuosobowym. Wymagania te są niezbędne do zaliczenia projektu i obejmują:

- Utworzenie nowego projektu (repozytorium) na platformie GitHub.
- Wykonanie co najmniej 5 commit'ów przez każdą z osób w grupie.
- Przeprowadzenie scenariusza pracy sekwencyjnej: jedna osoba tworzy gałąź, wykonuje commity i scala ją z gałęzią główną, a następnie druga osoba powtarza ten proces.
- Przeprowadzenie scenariusza pracy równoległej: obie osoby w tym samym czasie tworzą własne gałęzie, pracują na nich, a następnie obie scalają swoje zmiany z gałęzią główną.
- Celowe wygenerowanie i poprawne rozwiązanie co najmniej 6 konfliktów scalania (po 3 na osobę), które wystąpią podczas łączenia gałęzi.

1.3. Wymagania Dokumentacyjne

Ostatnim wymogiem jest przygotowanie kompletnej dokumentacji projektu, która składa się z:

- Dokumentacji technicznej kodu wygenerowanej automatycznie za pomocą narzędzia Doxygen.
- Niniejszej dokumentacji projektu, przygotowanej w systemie LaTeX, opisującej proces od analizy po implementację i wnioski.
- Dołączenia do dokumentacji LaTeX zrzutu ekranu przedstawiającego graf sieci (network) commit'ów z repozytorium GitHub, jako dowód na spełnienie wymagań procesowych.

2. Analiza problemu

Po określeniu ogólnych wymagań, niniejszy rozdział skupia się na szczegółowej analizie postawionych problemów. Zadanie zostało zdekomponowane na dwa główne, równoległe obszary: problem implementacji technicznej aplikacji w C++ oraz problem symulacji procesu pracy grupowej przy użyciu systemu kontroli wersji Git.

2.1. Wybór technologii i bibliotek

Język implementacji, C++, został narzucony w ramach zadania. Jest to język kompilowany, oferujący wysoką wydajność i bezpośrednią kontrolę nad zarządzaniem pamięcią, co jest kluczowe przy implementacji własnych dynamicznych struktur danych, takich jak drzewa. Projekt opiera się wyłącznie na standardowych bibliotekach C++ (STL), aby zapewnić maksymalną przenośność i zgodność. Do kluczowych bibliotek wykorzystanych w projekcie należą[3]:

- **¡iostream**: Niezbędna do obsługi podstawowego wejścia/wyjścia, czyli komunikacji z użytkownikiem w konsoli (operacje `std::cin` oraz `std::cout`).
- **fstream**: Używana do wszystkich operacji plikowych, zarówno tekstowych (`std::ifstream`, `std::ofstream`), jak i binarnych (z flagą `std::ios::binary`).
- **string**: Wykorzystywana do wygodnego zarządzania nazwami plików i innymi danymi tekstowymi.
- **limits**: Używana w funkcji `main` do walidacji danych wejściowych (czyszczenia bufora `std::cin`).

2.2. Architektura aplikacji

Kluczową decyzją analityczną było ścisłe rozdzielenie odpowiedzialności (zasada SRP - Single Responsibility Principle) poprzez podział programu na trzy odrębne moduły logiczne. Taka architektura była również wymogiem zadania (oddzielne pliki).

- **Klasa `BSTree`:** Stanowi "mózg"aplikacji. Jest odpowiedzialna wyłącznie za logikę struktury danych – przechowywanie wskaźnika na korzeń oraz implementację algorytmów dodawania, usuwania i przechodzenia drzewa.

[3]Pełna dokumentacja standardowych bibliotek C++ dostępna jest m.in. na portalu cppreference[3].

- **Klasa `FileManager`:** Pełni rolę warstwy dostępu do danych (Data Access Layer). Jej jedynym zadaniem jest serializacja i deserializacja obiektu drzewa do i z plików (tekstowych oraz binarnych).
- **Plik `main.cpp`:** Działa jako warstwa prezentacji (interfejs użytkownika). Nie przechowuje żadnej logiki biznesowej, a jedynie wyświetla menu, pobiera dane od użytkownika i wywołuje odpowiednie metody na obiektach `BSTree` i `FileManager`.

Taki podział nie tylko ułatwia zarządzanie kodem i testowanie, ale był fundamentalny dla spełnienia wymagań dotyczących pracy grupowej, pozwalając na równoległy rozwój różnych komponentów przez dwie osoby.

2.3. Analiza operacji na drzewie

Struktura danych BST opiera się na prywatnej, zagnieżdżonej strukturze `Wezel`, która przechowuje wartość `int` oraz dwa wskaźniki (`lewy` i `prawy`). Analiza algorytmów wykazała, że większość operacji (dodawanie, wyświetlanie, czyszczenie) jest stosunkowo prosta do zaimplementowania przy użyciu rekurencji. Największym wyzwaniem algorytmicznym w klasie `BSTree` jest operacja usuwania węzła. Musi ona poprawnie obsłużyć trzy fundamentalnie różne przypadki:

1. **Usuwanie liścia:** Węzeł nie ma dzieci. Wystarczy zwolnić pamięć i ustawić wskaźnik rodzica na `nullptr`.
2. **Usuwanie węzła z jednym dzieckiem:** Węzeł jest "pomijany"poprzez podłączenie jego jedynego dziecka bezpośrednio do jego rodzica.
3. **Usuwanie węzła z dwójką dzieci:** Najbardziej złożony przypadek, wymagający znalezienia następnika węzła (czyli węzła o najmniejszej wartości w prawym poddrzewie), skopiowania jego wartości do usuwanego węzła, a następnie rekurencyjnego usunięcia następnika (który z definicji będzie miał co najwyżej jedno dziecko).

2.4. Analiza operacji plikowych

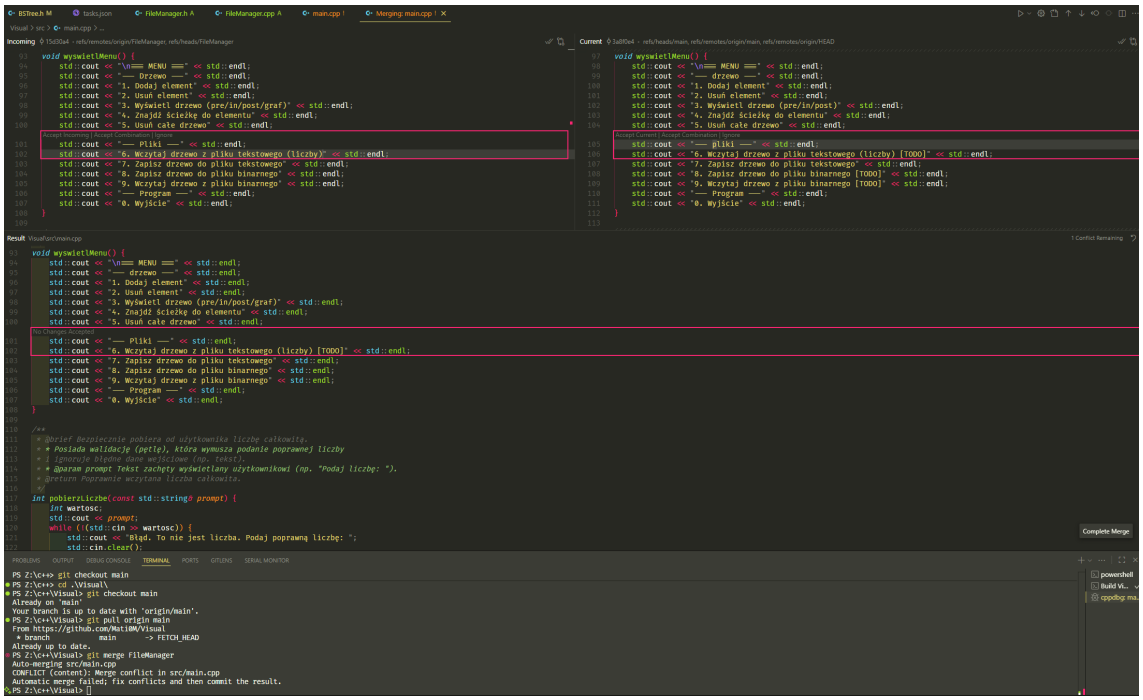
Wymagania dotyczące plików również niosły ze sobą różne poziomy skomplikowania.

- **Odczyt tekstowy:** Wczytanie pliku `liczby.txt` jest operacją prostą, polegającą na otwarciu strumienia `ifstream` i czytaniu kolejnych liczb całkowitych w pętli (operator `>>` inteligentnie pomija białe znaki).

- **Zapis binarny:** Zapis i odczyt binarny są znacznie bardziej złożone, ponieważ muszą odwzorować nie tylko dane, ale i strukturę (relacje) drzewa. Po analizie zdecydowano, że serializacja (zapis) będzie odbywać się poprzez przejście drzewa metodą **preorder** (Korzeń → Lewy → Prawy). Zapisanie wartości w tej kolejności gwarantuje, że podczas deserializacji (odczytu), proste wywoływanie publicznej funkcji `dodaj()` dla każdej wczytanej liczby odtworzy drzewo w dokładnie tej samej strukturze, w jakiej zostało zapisane.

2.5. Analiza procesu pracy grupowej (Git)

Drugi filar zadania stanowiła analiza i implementacja wymaganego procesu pracy z systemem Git i platformą GitHub[4]. Wymagania te wykraczały poza standardowe użycie, wymuszając symulację zaawansowanych scenariuszy, w tym pracy sekwencyjnej i równoległej na gałęziach. Największym wyzwaniem było celowe doprowadzenie do sześciu konfliktów scalania (merge conflicts). Konflikt taki występuje, gdy dwie osoby (na różnych gałęziach) zmodyfikują te same linie w tym samym pliku. Git nie jest wtedy w stanie automatycznie połączyć zmian i zatrzymuje proces, prosząc dewelopera o ręczne rozwiązanie problemu. Rysunek 2.1 (s. 10) przedstawia typowy widok takiego konfliktu w edytorze Visual Studio Code, gdzie programista musi wybrać, którą wersję kodu zachować (bieżącą czy przychodzącą), lub połączyć je ręcznie.



Rys. 2.1. Przykład konfliktu scalania (merge conflict) w edytorze VS Code

3. Projektowanie

Na podstawie przeprowadzonej analizy, w tym rozdziale przedstawiono konkretne decyzje projektowe dotyczące architektury oprogramowania. Projekt opiera się na trzech głównych filarach: klasie `BSTree` przechowującej logikę drzewa, klasie `FileManager` zarządzającej operacjami wejścia-wyjścia oraz module `main.cpp`, który pełni rolę interfejsu użytkownika.

3.1. Projektowanie klasy BSTree

Zaprojektowanie klasy `BSTree` było kluczowe dla całego projektu. Zdecydowano się na ścisłą hermetyzację (enkapsulację) danych. Jedynym polem składowym (atrybutem) klasy jest prywatny wskaźnik `Wezel* korzen`, który wskazuje na szczyt drzewa. Sama struktura `Wezel` została zdefiniowana jako prywatna, zagnieżdżona struktura wewnątrz klasy `BSTree`. Dzięki temu żaden komponent zewnętrzny (nawet `main.cpp`) nie ma wiedzy o tym, jak fizycznie zbudowany jest węzeł. Kluczową decyzją projektową było zapewnienie, że konstruktor struktury `Wezel` zawsze inicjalizuje wskaźniki `lewy` i `prawy` na wartość `nullptr`, co eliminuje ryzyko błędów pamięci. Zastosowano wzorzec projektowy "opakowania"(wrapper). Metody publiczne (np. `dodaj(int wartosc)`) są proste i stanowią jedynie interfejs dla użytkownika. Cała złożona logika rekurencyjna została zaimplementowana w odpowiadających im me-

todach prywatnych (np. `dodajHelper(Wezel* wezel, int wartosc)`), które operują bezpośrednio na wskaźnikach.

```
#ifndef BSTREE_H
#define BSTREE_H

#include <string>
#include <iosfwd>

class FileManager;
class BSTree {
friend class FileManager;

private:
    struct Wezel {
        int dane;
        Wezel* lewy;
        Wezel* prawy;
        Wezel(int wartosc) {
            dane = wartosc;
            lewy = nullptr;
            prawy = nullptr;
        }
    };
    Wezel* korzen;
    Wezel* dodajHelper(Wezel* wezel, int wartosc);
    Wezel* usunHelper(Wezel* wezel, int wartosc);
    Wezel* znajdzMin(Wezel* wezel);
    void wyczyscHelper(Wezel* wezel);
    void preorderHelper(Wezel* wezel);
    void inorderHelper(Wezel* wezel);
    void postorderHelper(Wezel* wezel);
    bool znajdzSciezkeHelper(Wezel* wezel, int wartosc);
    void zapiszHelper(Wezel* wezel, std::ostream& plik);
    void wyswietlGraficznieHelper(Wezel* wezel, std::string wciecie
   , bool czyPrawy);


public:
    BSTree();
    ~BSTree();
    void dodaj(int wartosc);
    void usun(int wartosc);
    void wyczysc();
    void znajdzSciezke(int wartosc);
```

```
42      void zapiszDoTekstowego(const std::string& nazwaPliku);
43      void wyswietl_preorder();
44      void wyswietl_inorder();
45      void wyswietl_postorder();
46      void wyswietlGraficznie();
47
48  };
49
50  #endif // BSTREE_H
```

Listing 1. Definicja interfejsu klasy BSTree (kod/BSTree.h)

Listing 1 (s. 11) przedstawia pełny plik nagłówkowy klasy `BSTree`, ukazujący podział na sekcje prywatną i publiczną.

3.2. Projektowanie klasy FileManager

Dla klasy `FileManager` przyjęto rolę klasy narzędziowej (utility class). Nie przechowuje ona żadnego stanu (nie posiada prywatnych pól składowych poza funkcjami). Jej jedynym celem jest realizacja zasady pojedynczej odpowiedzialności (SRP) poprzez odizolowanie logiki operacji plikowych od logiki struktury danych. Podczas projektowania tej klasy napotkano na istotny problem: metoda `zapiszBinarnie` musi mieć dostęp do prywatnej struktury `BSTree::Wezel` oraz prywatnego pola `BSTree::korzen`, aby móc rekurencyjnie przemierzać drzewo. Rozwiązaniem tego problemu było zadeklarowanie klasy `FileManager` jako "przyjaciela"klasy `BSTree` (`friend class FileManager;`). Jest to świadoma decyzja projektowa, która pozwala na ścisłą współpracę tych dwóch modułów przy jednoczesnym zachowaniu hermetyzacji `BSTree` przed resztą programu.

```
1  #ifndef FILEMANAGER_H
2  #define FILEMANAGER_H
3
4  #include <string>
5  #include "BSTree.h"
6  #include <iosfwd>
7  class FileManager {
8
9  private:
10     void zapiszBinarnieHelper(BSTree::Wezel* wezel, std::ostream&
       plik);
11
12 public:
13     void wczytajTekstowyDoDrzewa(BSTree& drzewo, const std::string&
       nazwaPliku);
```

```
14      void zapiszBinarnie(BSTree& drzewo, const std::string&
        nazwaPliku);
15      BSTree wczytajBinarnie(const std::string& nazwaPliku);
16
17  };
18
19  #endif // FILEMANAGER_H
```

Listing 2. Definicja interfejsu klasy FileManager (kod/FileManager.h)

Jak widać na listingu 2 (s. 12), klasa ta dołącza `BSTree.h` i implementuje prywatną metodę pomocniczą `zapiszBinarnieHelper` do obsługi rekurencyjnego zapisu binarnego. Projekt odczytu binarnego został oparty na wcześniejszej decyzji o zapisie w kolejności *preorder*.

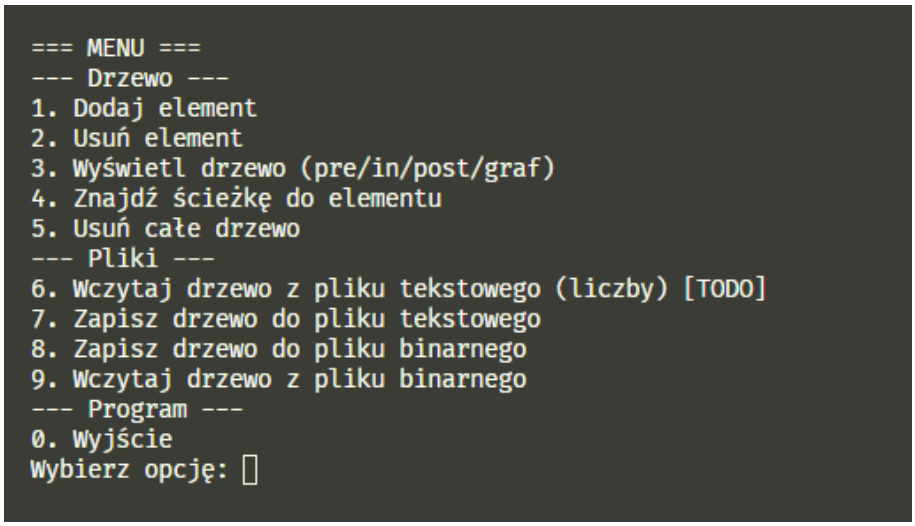
3.3. Projektowanie interfejsu użytkownika

Interfejs użytkownika (UI) został zaprojektowany jako prosta, ale robustna aplikacja konsolowa w pliku `main.cpp`. Głównym elementem jest nieskończona pętla `while(true)`, która w każdej iteracji wyświetla menu i czeka na wybór użytkownika. Do obsługi wyboru zastosowano instrukcję `switch-case`, która jest znacznie czytelniejsza niż wielokrotne instrukcje `if-else`. Logika każdej opcji menu (np. pytanie o liczbę, wywołanie metody) została wyekstrahowana do dedykowanych funkcji pomocniczych (np. `op_dodaj()`), co utrzymuje pętlę `switch` w czystej i czytelnej formie. Kluczowym elementem projektu UI jest również funkcja walidująca `pobierzLiczbe()`, która zabezpiecza program przed błędnymi danymi wejściowymi (np. wprowadzeniem tekstu zamiast liczby), czyszcząc strumień `std::cin` i prosząc o ponowne wprowadzenie danych[3].

Rysunek 3.1 (s. 14) prezentuje wygląd menu głównego programu po jego uruchomieniu.

Listing 3. Główna pętla programu i wywołania funkcji pomocniczych (kod/main.cpp)

Listing 3 (s. 13) ukazuje główną pętlę programu (funkcję `main()`) oraz sposób, w jaki funkcje pomocnicze są wywoływane w odpowiedzi na wybór użytkownika.



Rys. 3.1. Widok głównego menu aplikacji w konsoli

4. Implementacja

Faza implementacji polegała na przełożeniu założeń projektowych na działający kod w języku C++. Implementacja została podzielona na trzy główne komponenty, zgodnie z projektem: `BSTree.cpp` (logika drzewa), `FileManager.cpp` (operacje plikowe) oraz `main.cpp` (interfejs użytkownika).

4.1. Implementacja klasy BSTree

Implementacja klasy `BSTree` (listing 4, s. 14) zawiera definicje wszystkich metod zadeklarowanych w pliku nagłówkowym. Publiczne metody, takie jak `dodaj()` czy `usun()`, działają jako proste "opakowania"(wrappers), które wywołują swoje rekurencyjne, prywatne odpowiedniki (np. `dodajHelper()`), przekazując im wskaźnik na korzeń drzewa.

Najbardziej złożonym algorytmem jest `usunHelper(Wezel* wezel, int wartosc)`. Implementacja ta poprawnie obsługuje wszystkie trzy przypadki usuwania węzła, w tym najbardziej skomplikowany (usuwanie węzła z dwójką dzieci) poprzez znalezienie jego następnika inorder (metoda `znajdzMin()`) i skopiowanie jego danych.

```
1 #include "BSTree.h"
2 #include <iostream>
3 #include <fstream>
4 #include <string>
5 BSTree::BSTree() {
6     korzen = nullptr;
7 }
```

```
BSTree::~BSTree() {
    wyczysc();
}
void BSTree::dodaj(int wartosc) {
    korzen = dodajHelper(korzen, wartosc);
}
void BSTree::usun(int wartosc) {
    korzen = usunHelper(korzen, wartosc);
}
void BSTree::wyczysc() {
    wyczyscHelper(korzen);
    korzen = nullptr;
}
void BSTree::znajdzSciezke(int wartosc) {
    std::cout << "Sciezka do " << wartosc << ": ";
    if (znajdzSciezkeHelper(korzen, wartosc) == false) {
        std::cout << "Nie znaleziono elementu.";
    }
    std::cout << std::endl;
}
void BSTree::zapiszDoTekstowego(const std::string& nazwaPliku) {
    std::ofstream plikWyjsciowy(nazwaPliku);

    if (!plikWyjsciowy.is_open()) {
        std::cout << "Blad: Nie mozna otworzyc pliku do zapisu!" <<
    std::endl;
        return;
    }

    zapiszHelper(korzen, plikWyjsciowy);

    plikWyjsciowy.close();
    std::cout << "Zapisano drzewo do pliku: " << nazwaPliku << std
   ::endl;
}
void BSTree::wyswietl_preorder() {
    std::cout << "Preorder: [ ";
    preorderHelper(korzen);
    std::cout << "]" << std::endl;
}
void BSTree::wyswietl_inorder() {
    std::cout << "Inorder: [ ";
    inorderHelper(korzen);
    std::cout << "]" << std::endl;
}
```

```
51 void BSTree::wyswietl_postorder() {
52     std::cout << "Postorder: [ ";
53     postorderHelper(korzen);
54     std::cout << "]" << std::endl;
55 }
56 void BSTree::wyswietlGraficznie() {
57     std::cout << "--- Struktura drzewa (obrocone o 90 stopni) ---"
       << std::endl;
58     wyswietlGraficznieHelper(korzen, "", false);
59     std::cout << "-----------------------------------------------"
       << std::endl;
60 }
61 BSTree::Wezel* BSTree::dodajHelper(Wezel* wezel, int wartosc) {
62     if (wezel == nullptr) {
63         return new Wezel(wartosc);
64     }
65 
66     if (wartosc < wezel->dane) {
67         wezel->lewy = dodajHelper(wezel->lewy, wartosc);
68     } else if (wartosc > wezel->dane) {
69         wezel->prawy = dodajHelper(wezel->prawy, wartosc);
70     }
71 
72     return wezel;
73 }
74 BSTree::Wezel* BSTree::usunHelper(Wezel* wezel, int wartosc) {
75     if (wezel == nullptr) {
76         std::cout << "Nie znaleziono elementu " << wartosc << " do
       usuniecia." << std::endl;
77         return nullptr;
78     }
79 
80     if (wartosc < wezel->dane) {
81         wezel->lewy = usunHelper(wezel->lewy, wartosc);
82     } else if (wartosc > wezel->dane) {
83         wezel->prawy = usunHelper(wezel->prawy, wartosc);
84     }
85     else {
86         if (wezel->lewy == nullptr && wezel->prawy == nullptr) {
87             delete wezel;
88             return nullptr;
89         }
90         else if (wezel->lewy == nullptr) {
91             Wezel* temp = wezel->prawy;
92             delete wezel;
```

```
93              return temp;
94          }
95          else if (wezel->prawy == nullptr) {
96              Wezel* temp = wezel->lewy;
97              delete wezel;
98              return temp;
99          }
100         else {
101             Wezel* nastepnik = znajdzMin(wezel->prawy);
102             wezel->dane = nastepnik->dane;
103             wezel->prawy = usunHelper(wezel->prawy, nastepnik->dane
    );
104         }
105     }
106     return wezel;
107 }
108 BSTree::Wezel* BSTree::znajdzMin(Wezel* wezel) {
109     Wezel* obecny = wezel;
110     while (obecny != nullptr && obecny->lewy != nullptr) {
111         obecny = obecny->lewy;
112     }
113     return obecny;
114 }
115 void BSTree::wyczyscHelper(Wezel* wezel) {
116     if (wezel == nullptr) {
117         return;
118     }
119     wyczyscHelper(wezel->lewy);
120     wyczyscHelper(wezel->prawy);
121     delete wezel;
122 }
123 void BSTree::preorderHelper(Wezel* wezel) {
124     if (wezel == nullptr) return;
125     std::cout << wezel->dane << " ";
126     preorderHelper(wezel->lewy);
127     preorderHelper(wezel->prawy);
128 }
129 void BSTree::inorderHelper(Wezel* wezel) {
130     if (wezel == nullptr) return;
131     inorderHelper(wezel->lewy);
132     std::cout << wezel->dane << " ";
133     inorderHelper(wezel->prawy);
134 }
135 void BSTree::postorderHelper(Wezel* wezel) {
136     if (wezel == nullptr) return;
```

```
137     postorderHelper(wezel->lewy);
138     postorderHelper(wezel->prawy);
139     std::cout << wezel->dane << " ";
140 }
141 bool BSTree::znajdzSciezkeHelper(Wezel* wezel, int wartosc) {
142     if (wezel == nullptr) {
143         return false;
144     }
145 
146     std::cout << wezel->dane;
147 
148     if (wezel->dane == wartosc) {
149         std::cout << " (Znaleziono)";
150         return true;
151     }
152 
153     std::cout << " -> ";
154     if (wartosc < wezel->dane) {
155         return znajdzSciezkeHelper(wezel->lewy, wartosc);
156     } else {
157         return znajdzSciezkeHelper(wezel->prawy, wartosc);
158     }
159 }
160 void BSTree::zapiszHelper(Wezel* wezel, std::ostream& plik) {
161     if (wezel == nullptr) {
162         return;
163     }
164     zapiszHelper(wezel->lewy, plik);
165     plik << wezel->dane << "\n";
166     zapiszHelper(wezel->prawy, plik);
167 }
168 void BSTree::wyswietlGraficznieHelper(Wezel* wezel, std::string
    wciecie, bool czyPrawy) {
169     if (wezel == nullptr) {
170         return;
171     }
172 
173     wyswietlGraficznieHelper(wezel->prawy, wciecie + "    ", true);
174 
175     std::cout << wciecie;
176     if (czyPrawy) {
177         std::cout << "/---";
178     } else {
179         std::cout << "\\---";
180     }
```

```
181     std::cout << wezel->dane << std::endl;
182 
183     wyswietlGraficznieHelper(wezel->lewy, wciecie + "    ", false);
184 }
```

Listing 4. Implementacja logiki drzewa BST (kod/BSTree.cpp)

Wszystkie metody wyświetlające (preorder, inorder, postorder) są również zaimplementowane rekurencyjnie. Rysunek 4.1 (s. 20) prezentuje wynik działania metody `wyswietlGraficznie()`, która wykorzystuje odwróconą kolejność *inorder* do renderowania struktury drzewa w konsoli.

4.2. Implementacja klasy FileManager

Implementacja klasy `FileManager` (listing 5, s. 19) zawiera logikę operacji wejścia/wyjścia. Metoda `wczytajTekstowyDoDrzewa()` wykorzystuje prostą pętlę `while (plik >> liczba)` do odczytu danych z pliku tekstowego.

Zgodnie z projektem, zapis binarny jest realizowany w kolejności *preorder*. Prywatna funkcja `zapiszBinarnieHelper()` rekurencyjnie zapisuje dane węzła (`plik.write()`), a następnie wywołuje samą siebie dla lewego i prawego dziecka. Kluczowe jest tu użycie `sizeof(int)`, aby zapewnić zapisanie pełnych 4 bajtów dla każdej liczby.

Należy zwrócić uwagę na implementację odczytu binarnego (`wczytajBinarnie`). Zgodnie z tym, co widać na listingu 2 (s. 12), funkcja ta zwraca nowy obiekt `BSTree`. Działa to w oparciu o logikę zapisu *preorder*, która gwarantuje, że proste dodawanie wczytanych elementów do nowego drzewa odtworzy jego pierwotną strukturę.

```
1  #include "FileManager.h"
2  #include <fstream>
3  #include <iostream>
4  void FileManager::wczytajTekstowyDoDrzewa(BSTree& drzewo, const std
       ::string& nazwaPliku) {
5      std::ifstream plikWejsciowy(nazwaPliku);
6  
7      if (!plikWejsciowy.is_open()) {
8          std::cout << "Blad: Nie mozna otworzyc pliku: " <<
       nazwaPliku << std::endl;
9          return;
10     }
11 
12     int liczba;
13     while (plikWejsciowy >> liczba) {
14         drzewo.dodaj(liczba);
15     }
```

```
=== MENU ===
--- Drzewo ---
1. Dodaj element
2. Usuń element
3. Wyświetl drzewo (pre/in/post/graf)
4. Znajdź ścieżkę do elementu
5. Usuń całe drzewo
--- Pliki ---
6. Wczytaj drzewo z pliku tekstowego (liczby) [TODO]
7. Zapisz drzewo do pliku tekstowego
8. Zapisz drzewo do pliku binarnego
9. Wczytaj drzewo z pliku binarnego
--- Program ---
0. Wyjście
Wybierz opcję: 6
Podaj nazwę pliku .txt do wczytania: test.txt
Wczytano liczby z pliku test.txt.

Naciśnij Enter, aby kontynuować...

=== MENU ===
--- Drzewo ---
1. Dodaj element
2. Usuń element
3. Wyświetl drzewo (pre/in/post/graf)
4. Znajdź ścieżkę do elementu
5. Usuń całe drzewo
--- Pliki ---
6. Wczytaj drzewo z pliku tekstowego (liczby) [TODO]
7. Zapisz drzewo do pliku tekstowego
8. Zapisz drzewo do pliku binarnego
9. Wczytaj drzewo z pliku binarnego
--- Program ---
0. Wyjście
Wybierz opcję: 3
Wybierz metodę wyświetlania:
1. Preorder
2. Inorder (posortowane)
3. Postorder
4. Graficznie (struktura)
Wybór: 4
--- Struktura drzewa (obrocone o 90 stopni) ---
        /---80
    /---70
        \---60
\---50
        /---40
    \---30
        \---20
------------------------------------------------

Naciśnij Enter, aby kontynuować...
```

Rys. 4.1. Wynik działania opcji "Wyświetl drzewo graficznie"

```
16
17 void FileManager::zapiszBinarnie(BSTree& drzewo, const std::string&
       nazwaPliku) {
18     std::ofstream plikWyjsciowy(nazwaPliku, std::ios::binary);
19
20     if (!plikWyjsciowy.is_open()) {
21         std::cout << "Blad: Nie mozna otworzyc pliku binarnego: "
    << nazwaPliku << std::endl;
22         return;
23     }
24
25     zapiszBinarnieHelper(drzewo.korzen, plikWyjsciowy);
26
27     plikWyjsciowy.close();
28     std::cout << "Zapisano drzewo binarnie do " << nazwaPliku << ".
    " << std::endl;
29 }
30 void FileManager::zapiszBinarnieHelper(BSTree::Wezel* wezel, std::
    ostream& plik) {
31     if (wezel == nullptr) {
32         return;
33     }
34
35     plik.write( (char*)&(wezel->dane), sizeof(int) );
36     zapiszBinarnieHelper(wezel->lewy, plik);
37     zapiszBinarnieHelper(wezel->prawy, plik);
38 }
39 BSTree FileManager::wczytajBinarnie(const std::string& nazwaPliku)
    {
40
41     BSTree noweDrzewo;
42
43     std::ifstream plikWejsciowy(nazwaPliku, std::ios::binary);
44
45     if (!plikWejsciowy.is_open()) {
46         std::cout << "Blad: Nie mozna otworzyc pliku binarnego: "
    << nazwaPliku << std::endl;
47         return noweDrzewo;
48     }
49
50     int liczba;
51
52     while ( plikWejsciowy.read( (char*)&liczba, sizeof(int)) ) {
53         noweDrzewo.dodaj(liczba);
54     }
```

```
55
56      plikWejsciowy.close();
57      std::cout << "Wczytano drzewo z pliku binarnego " << nazwaPliku
        << "." << std::endl;
58
59      return noweDrzewo;
60  }
```

Listing 5. Implementacja menedżera plików (kod/FileManager.cpp)

4.3. Implementacja programu głównego

Plik `main.cpp` (listing 6, s. 22) zawiera implementację interfejsu użytkownika. Inicjalizuje obiekty `mojeDrzewo` i `managerPlikow`, a następnie uruchamia pętlę `while(true)`. Do obsługi menu wykorzystano instrukcję `switch-case`.

Aby utrzymać czystość kodu w głównej pętli, logika każdej opcji menu została wyodrębniona do osobnej funkcji pomocniczej (np. `op_dodaj(BSTree& drzewo)`). Funkcje te pobierają niezbędne dane od użytkownika (np. numer lub nazwę pliku) i wywołują odpowiednie metody na przekazanych obiektach.

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4
5  int main(int argc, char** argv) {
6
7    ofstream plik("strona.html");
8    if(!plik)
9      cout<<"blad zapisu pliku";
10   else
11     {
12     plik<<"<html>";
13     plik<<"<head><title>Moja pierwsza strona www</title></head>";
14     plik<<"<body>Strona WWW</body>";
15     plik<<"</html>";
16
17     cout<<"Wygenerowana strona";
18     }
19     plik.close();
20
21
22     return 0;
```

```
23 }
```

Listing 6. Finalna implementacja programu głównego (kod/main.cpp)

5. Wnioski

Realizacja niniejszego projektu zakończyła się pełnym sukcesem. Udało się zaimplementować wszystkie wymogi funkcjonalne aplikacji, tworząc stabilny program do zarządzania strukturą binarnego drzewa poszukiwań. Co jednak ważniejsze, osiągnięto główny cel dydaktyczny zadania: przećwiczono w praktyce zaawansowane scenariusze pracy grupowej z wykorzystaniem systemu kontroli wersji Git oraz platformy GitHub[4].

5.1. Wrażenia z implementacji programu

Z perspektywy programistycznej, projekt był doskonałym ćwiczeniem z zakresu dynamicznego zarządzania pamięcią i implementacji fundamentalnych struktur danych. Największym wyzwaniem okazał się algorytm usuwania węzła z drzewa, który wymagał precyzyjnego zarządzania wskaźnikami w trzech różnych przypadkach. Również implementacja zapisu i odczytu binarnego, przy jednoczesnym poprawnym zarządzaniu cyklem życia obiektów w pamięci, była cenną lekcją.

Podział aplikacji na trzy odrębne moduły (`BSTree`, `FileManager`, `main`) okazał się kluczowy. Ułatwiło to nie tylko samą implementację i testowanie, ale było fundamentem, który umożliwił efektywną pracę równoległą w systemie Git.

5.2. Doświadczenia z pracy grupowej w GitHub

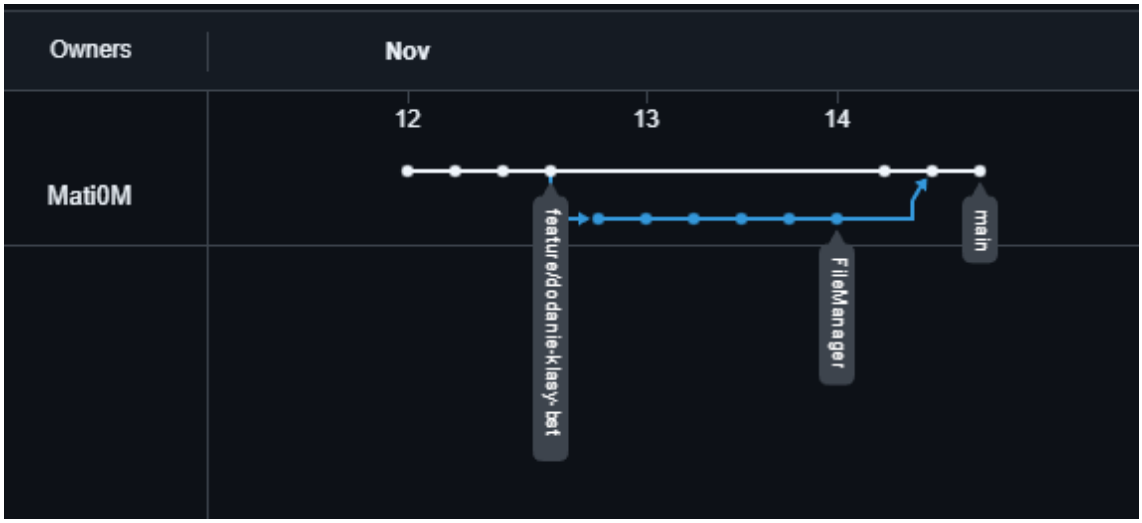
Główną wartością dodaną projektu była intensywna praca z systemem Git, która wykraczała poza proste wysyłanie zmian. Wymóg realizacji scenariuszy pracy sekwencyjnej, równoległej oraz celowego generowania konfliktów pozwolił w pełni zrozumieć, dlaczego Git jest standardem w branży IT.

Obaj autorzy projektu pozytywnie oceniają to doświadczenie. Jeden z autorów (Rafał), który miał już wcześniej styczność z systemem, ponownie potwierdził jego użyteczność: *„od dawna mam styczność z git i uważam że jest to świetne roziwązanie które ułatwia pracę”*. Dla drugiego autora (Mateusz) była to jedna z pierwszych okazji do tak intensywnej pracy z gałęziami. Jego odczucia są podobne: *„po pierwszych wrażaniach z git i gałęziami uważam to za świetne rozwiązanie ułatwiającer pracę”*.

Najbardziej pouczającym elementem było celowe generowanie i rozwiązywanie konfliktów scalania. Doświadczenie to pokazało, w jaki sposób Git chroni integralność kodu i wymusza na programistach komunikację oraz świadome podejmo-

wanie decyzji w momencie łączenia zmian (co zostało zilustrowane w rozdziale 2 na rysunku 2.1, s. 10).

Dowodem na przeprowadzenie tych operacji jest graf sieci commitów, pobrany z repozytorium GitHub, przedstawiony na rysunku 5.1 (s. 25).



Rys. 5.1. Graf sieci commitów z repozytorium projektu na GitHub (stan na 14.11.2025, mógł ulec zmianom)

5.3. Wnioski końcowe

Ostatnim etapem projektu było przygotowanie dokumentacji. Zastosowanie narzędzia Doxygen[5] (rysunek 5.2, s. 27) pokazało, jak dużą wartość ma konsekwentne stosowanie ustandaryzowanych komentarzy, które pozwalają na automatyczne generowanie profesjonalnej dokumentacji technicznej.

Sam proces tworzenia niniejszej dokumentacji w systemie LaTeX[6] był kolejnym wyzwaniem. Największą trudnością okazało się zintegrowanie plików kodu źródłowego, które były zapisane w starszym kodowaniu (Windows-1250), z głównym dokumentem w kodowaniu UTF-8. Problem ten został ostatecznie rozwiązany poprzez zmianę kompilatora w Overleaf na `XeLaTeX`, który natywnie obsługuje różne kodowania i nowoczesne czcionki.

Podsumowując, projekt był cennym doświadczeniem, które połączyło klasyczny problem algorytmiczny (implementacja BST) z nowoczesnymi, niezbędnymi w pracy dewelopera narzędziami (Git, Doxygen, LaTeX).

Bibliografia

[1] Wikipedia. *Binarne drzewo poszukiwań*. URL: https://pl.wikipedia.org/wiki/Binarne_drzewo_poszukiwa%C5%84 (term. wiz. 14. 11. 2025).

[2] David Galles. *Data Structure Visualization: Binary Search Tree*. URL: https://www.cs.usfca.edu/~galles/visualization/BST.html (term. wiz. 14. 11. 2025).

[3] *cppreference.com - Standard library header ¡fstream¿*. URL: https://en.cppreference.com/w/cpp/header/fstream (term. wiz. 14. 11. 2025).

[4] *GitHub: Where the world builds software*. URL: https://github.com/ (term. wiz. 14. 11. 2025).

[5] *Doxygen: Source code documentation generator*. URL: https://www.doxygen.nl/ (term. wiz. 14. 11. 2025).

[6] *The LaTeX Project*. URL: https://www.latex-project.org/ (term. wiz. 14. 11. 2025).

Rys. 5.2. Fragment dokumentacji technicznej wygenerowanej przez Doxygen (PDF)

Spis rysunków

[4]Interaktywna animacja procesu budowania drzewa BST jest dostępna online[2].

Spis tabel

Spis listingów